TUHH
Hamburg University of Technology

Project Work

Marcel Gehrke

A Frama-C Plug-In for Finding Equal-Valued Expressions Using Dataflow Analysis

January 30, 2014

supervised by:
Prof. Dr. Sibylle Schupp
Sven Mattsen

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg

TUHH STS
Software
Technology
Systems

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich die vorliegende Projektarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Hamburg, den 30. Januar 2014

Marcel Gehrke

Contents

1 Introduction **1**

2 Dataflow Analysis **3**
2.1 Control Flow Graph . . . 3
2.2 Dataflow Analysis in General . . . 4
2.2.1 Example . . . 4
2.2.2 General . . . 5
2.2.3 Formal Definition . . . 6
2.2.4 Worklist Algorithm . . . 7
2.3 Dataflow Analysis to Find Equivalent Warnings . . . 8
2.3.1 Example . . . 8
2.3.2 Formal Definition . . . 10

3 Plug-In to Remove Redundant Warnings **13**
3.1 Frama-C . . . 13
3.2 Implementation of the Plug-In . . . 13
3.3 Evaluation . . . 15
3.3.1 Example . . . 15
3.3.2 Properties of the Tool . . . 17
3.3.3 Evaluation Against a Built-In Function . . . 18
3.3.4 Evaluation of Different Combinations . . . 20

4 Future Work & Conclusion **23**
4.1 Future Work . . . 23
4.2 Conclusion . . . 24

Bibliography **25**

Appendix **27**

List of Figures

2.1 CFG of the Example Code . . . 3
2.2 Dataflow Equations of a Must Analysis . . . 6
2.3 Worklist Algorithm [3] . . . 7
2.4 Forwards Dataflow Equations to Find Equivalent Warnings [6] . . . 11

List of Tables

2.1 DFA Result of the Must Analysis Example . . . 5
2.2 DFA Result of the May Analysis Example . . . 5
2.3 Possible DFA Classification . . . 6
2.4 DFA Result of the Warning Example . . . 9
2.5 DFA Result of the Warning Example without the Reassignment . . . 10
2.6 Dataflow Equations Applied to Our Example . . . 12

3.1 Example Computation using the Plug-In . . . 16
3.2 Combinations of Statements . . . 20
3.3 Combinations of Statements . . . 21

Listings

1.1 Short C Example . . . 1
1.2 Short C Example with All Warnings . . . 1
1.3 Short C Example with All Needed Warnings . . . 2

2.1 Example Code . . . 3
2.2 Example Code with Redundant Warnings . . . 8
2.3 Example Code without Redundant Warnings . . . 9

3.1 Array Example Code . . . 17
3.2 Example Code for Combing Branches . . . 18
3.3 Too Many Warnings Example Code . . . 19
3.4 Combination of Statements . . . 22

4.1 Path Sensitivity Example . . . 23
4.2 Assignment Tracking Example . . . 23

3 combinePredecessors . . . 27

4 doInstr 27

1 Introduction

Frama-C is a framework to statically analyze C code. One possible way of using Frama-C is to try to automatically verify C code. The value analysis is a Frama-C plug-in that tries to verify C code. However, in the progress of verifying the C code, one can encounter a number of warnings. Reviewing them all by hand can be quite cumbersome. Therefore, we would like to reduce the number of warnings displayed to the Frama-C user. To understand how we can reduce the number of warnings displayed, let us have a look at a simple example.

```
1  void f1(int x) {
2    int h = 1/x;
3    h = 1/x;
4  }
```

Listing 1.1: Short C Example

In Lst. 1.1, we see a short C program. The program simply calculates the expression `1/x` twice. However, since in the code the variable x is passed as an argument, Frama-C does not know the value of the variable by looking at the function. The variable `x` could have the value `0`. Hence, it could be a division-by-zero. That is not allowed for integers in C and yields a warning in the value analysis of Frama-C.

```
1  void f1(int x) {
2    /*@ assert Value: division_by_zero: x ≠ 0; */
3    int h = 1/x;
4    /*@ assert Value: division_by_zero: x ≠ 0; */
5    h = 1/x;
6  }
```

Listing 1.2: Short C Example with All Warnings

Running the value analysis of Frama-C on the code in Lst. 1.1 gives us exactly what we expect. The value analysis tries to verify the code. Nonetheless, it cannot guarantee that x is not equal to 0 since the range of the variable is not restricted. Thus, Frama-C annotates the code as depicted in Lst. 1.2. The assertions in line 2 and 4 originate from warnings.

The assertion in line 2 and line 4 are redundant. They are redundant since the warnings are thrown by equal-valued expressions. In this case, the warnings are thrown by the same expression without any of its variables being reassigned in between the warnings. Further, when the user reviews these warnings, it is sufficient for him to look at either of these redundant warnings. Therefore, Frama-C could display only the annotation in line 2. In Lst. 1.3 we see the version where the redundant warning in line 4 is removed.

One idea to reduce the number of warnings displayed is to group redundant warnings. Redundant warnings are warnings thrown by equal-valued expressions. T. Muske, A. Baid and T. Sanas present an approach to group redundant warnings in their paper

"Review Efforts Reduction by Partitioning of Static Analysis Warnings" [6]. The idea is to track equal-valued expression in case they throw a warning. Equal-valued expressions evaluate to the same result during run-time. Warnings of equal-valued expressions can be grouped together in a redundant set. Showing just one warning of each redundant set can reduce the overall number of displayed warnings. However, Frama-C cannot display just any warning of the redundant set but has to display the most general warning. The most general warning has a path in the code to all other warnings of the redundant set. In the analysis we apply, which we explain in depth in Chapter 2, we always have a most general warning or at least a combination of warnings that make up the most general warning. Displaying the most general warning provides all crucial information to the user.

```
1  void f1(int x) {
2    /*@ assert Value: division_by_zero: x ≠ 0; */
3    int h = 1/x;
4    h = 1/x;
5  }
```

Listing 1.3: Short C Example with All Needed Warnings

Let us now check that the user still has all the necessary information. There are two possibilities for the most general warning. Either it is a false positive or the fault can actually occur. In our example, if the warning is a false positive, the second warning will be one as well. In that case, the first warning is sufficient for the user. False positive in this case means that Frama-C throws a warning of a fault that will not occur.

If the warning's fault occurs, the user also has to look at the other warnings of the set. In our case, both warnings would lead to a fault. Fixing the warning in line 2 would result in both warnings to disappear. Another possibility is that only the first warning's fault occurs and the second is a false positive. An example of that would be a branch that restricts the valuation of `x` to be not equal to zero. Then, the second warning would be a false positive and could be removed even though the first warning could lead to a fault. Hence, the user still has all the crucial information seeing only the most general warning.

To sum it up, the described algorithm can reduce the effort of reviewing the warnings. The user has to check whether a warning is a false positive or can lead to an actual fault. As mentioned before, showing the most general warning to be a false positive the user does not have to look at the other warnings of the redundant set. If the fault of the warning can occur, the user will also have to have a look at the warnings that are redundant individually. Using this algorithm, Frama-C does not only show less warnings to the user, while still providing him with all the information he had before, but also reduce the effort of the user to review warnings.

In the following, we will first take a look at dataflow analysis in Chapter 2. A dataflow analysis is the method we apply to find redundant warnings as proposed by Muske et al. in [6]. After this theoretical part, we will have a look at the implemented tool and how it performs in Chapter 3. Lastly in Chapter 4, we have a look at possible improvements.

2 Dataflow Analysis

A dataflow analysis (DFA) is a class of methods to statically analyze code. Using a DFA, one tries to gain information about the state at a certain point in the program. To be able to gain the information, we need to know how the data can possibly flow through the program. A control-flow graph (CFG) of a program provides the details how the data can flow in the code. Knowing how data can flow, we can track information of interest. For our purpose of finding equivalent warnings to group them, warnings are the information of interest. To be more precise we want to know if a warning at a certain point in the program is redundant to a warning thrown before.

In this chapter, we will first have a look at what exactly a CFG is. Afterwards, we use the CFG in a DFA and have a closer look at DFA in general to know how we can track information of interest. Knowing how a DFA can look like, we will apply it to our case and track warnings.

2.1 Control Flow Graph

In this section we define what a CFG is and have a look at a little example of a CFG. The example will be used throughout this chapter.

```
1  void f1(int x, int y) {
2    int h = 1/x;[1]
3    while(y)[2] {
4        h = 1/x;[3]
5        x = y;[4]
6    }
7    h = 1/x;[5]
8    h = 1/x;[6]
9  }
```

Listing 2.1: Example Code

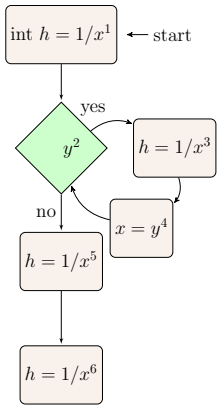

Figure 2.1: CFG of the Example Code

Fig. 2.1 shows us how a CFG for the code in Lst. 2.1 looks. A CFG is a directed graph. The edges show how we can traverse through the code. The nodes of the graph correspond to blocks in the code. Each block consists of one statement. Thus, there cannot be any kind of branching in a block. Branching can be achieved by ,e.g., `if` conditions or `while loops`.

To see how the program code and the CFG correlate, we start by mapping it. First of all, we have the first statement of the code `int h = 1/x;`. The first statement is followed by a `while loop` condition and therefore, we have two branches in the CFG.

If we enter the `while loop`, we execute the statements in line 4 and 5. Afterwards, we combine the branches again. The statement in line 7 is reachable in case the condition of the `while loop` evaluates to false. The rest of the CFG is straightforward. There are no more statements that influence the flow. The remaining statements are now executed sequentially. Further, each block in the CFG has a labeling number to identify it.

Having the CFG, we now have a graph that shows us every possible way through the program. To be more precise, we can track data knowing every possible way it can take through the code. To know how we can track data, we take a look at DFA in general.

2.2 Dataflow Analysis in General

In a DFA, we want to have a notion of what the data can look like at certain points in the program. To familiarize oneself with how and what information we can track, we start by taking a look at a little example of a DFA. Let us say in our DFA, we want to know if an expression must have been calculated already in the current point of the program. An expression is calculated, if an expression has been calculated before and no variable of the expression has been newly assigned on any path reaching the current point. We can apply the DFA to the CFG in Fig. 2.1.

2.2.1 Example

In the beginning, no expressions are calculated. Every time an expression is calculated, we start to track it by adding it to a set of active expressions. If one of the variables of an active expression is reassigned, we remove that expression from the set and stop tracking it. Our DFA goes through the CFG as follows:

- First block: The expression $1/x$ is calculated. Thus, $1/x$ is now active. We do not reassign any variables of active expressions. Hence, all active expressions stay active.
- Third block: We calculate the expression $1/x$ and set it active.
- Fourth block: The statement consists of a reassignment of x. $1/x$ is active coming from the third block. We reassign a variable of an active expression. Therefore, we have to remove it from the set of active expressions.
- Second block: The condition of the `while loop` has two predecessors, in this case, the first and the fourth block. Since the expressions must be active on every path and $1/x$ is only active on one path, there are no active expressions reaching the second block.
- Fifth block: In the statement, we calculate the expression $1/x$ and set it active once more.
- Sixth block: The last statement also consists of the calculation of the expression $1/x$.

The code in Lst. 2.1 calculates the expression $1/x$ four times. Tab. 2.1 depicts the result of the DFA, i.e., what expression is active and already calculated in a block. From the result, we can see that the expression is only active once, when it is calculated. In the fifth block, we set $1/x$ active. That information reaches the sixth block in which $1/x$ is calculated again. In the sixth block, we do not have to recalculate the expression, we know that the expression is active. On all the other occurrences, we have to calculate $1/x$.

Block	1	3	4	2	5	6
In	$\emptyset$	$\emptyset$	$\{1/x\}$	$\emptyset$	$\emptyset$	$\{1/x\}$
Out	$\{1/x\}$	$\{1/x\}$	$\emptyset$	$\emptyset$	$\{1/x\}$	$\{1/x\}$
Already Calculated						✓

Table 2.1: DFA Result of the Must Analysis Example

2.2.2 General

In general we have a few choices when it comes to a DFA. The class of DFAs consists of have a *must* or a *may* or a forwards or backwards analysis. Due to branching, we have to decide what to do when we combine branches again. That decision results in either a *must* or *may analysis*. The difference between a *must* or a *may* analysis is how to combine predecessors. In a *must* analysis, we combine the sets of data with an intersection. Hence, in a *must* analysis, information must be true on all paths to still hold after combining predecessors. For a may analysis, the predecessors are combined with a union. A *may* analysis only requires that the information is true in one of these paths. Since we cannot be certain that the program will take the path the information is true on, the information may be true after the combination, thus, a *may* analysis.

Block	1	3	4	2	5	6
In	$\emptyset$	$\{1/x\}$	$\{1/x\}$	$\{1/x\}$	$\{1/x\}$	$\{1/x\}$
Out	$\{1/x\}$	$\{1/x\}$	$\emptyset$	$\{1/x\}$	$\{1/x\}$	$\{1/x\}$
Already Calculated		✓			✓	✓

Table 2.2: DFA Result of the May Analysis Example

For a *may* analysis, the result of our example is different. In a *may* analysis an expression just needs to be active on one path. In the first block $1/x$ is calculated and set active. As before, the expression is not active after executing the `while loop`. However, after combining the branches, in the second block the expression is active since it is active on the path from the first statement. Now, we propagate to the third and fifth block that $1/x$ is active. Tab. 2.2 shows the result of the *may* analysis. In nearly every block the expression is active and may be calculated. Thus, the expression $1/x$ may be calculated already at every occasion but the first.

	May	*Must*
Forwards		
Backwards		

Table 2.3: Possible DFA Classification

The other two classification criteria are forward and backward analyses. The difference is in which direction we go through the CFG. To be more precise, we go either from the unique entry to the exit of the program or the other way around, i.e., from the exit to the entry. Combining these possible analyses we have four basic types of DFA. Tab. 2.3 depicts the four basic types of DFA.

2.2.3 Formal Definition

A DFA is in general defined by its dataflow equations [4]. The dataflow equations for a forwards analysis look as follows. For each block n in the CFG:

$$\begin{aligned} out_n &= transfer_n(in_n) \\ in_n &= join_{p \in pred_n}(out_n) \end{aligned}$$

The `transfer` function defines what we do in a block. Hence, the `transfer` function specifies what kind of information we generate and what information we kill.

$$\begin{aligned} In_n &= \begin{cases} \emptyset & n \text{ is the entry block} \\ \bigcap_{p \in predecessors(n)} Out_n & \text{otherwise} \end{cases} \\ Out_n &= (Gen_n + In_n) - Kill_n(Gen_n + In_n) \\ Gen_n &= \begin{cases} \{e\} & \text{if } n \text{ computes} \\ \emptyset & \text{otherwise} \end{cases} \\ Kill_n(X) &= \begin{cases} killInfo(X, n) & n \text{ modifies at least one variable} \\ \emptyset & \text{no variable is modified by } n \end{cases} \\ killInfo(X, n) &= \{e \in X \mid (usedVars(e) \cap modifiedVars(n)) \neq \emptyset\} \\ usedVars(e) &= \text{r-values from expression } e \\ modifiedVars(n) &= \text{l-values from program statement } n \end{aligned}$$

Figure 2.2: Dataflow Equations of a Must Analysis

To differentiate between a *may* and *must* analysis, the `join` function is either the intersection or union. For a backwards analysis, `out` and `in` are exchanged. Further, we `join` the successors and not the predecessors.

Fig. 2.2 shows the dataflow equations of the *must* analysis from subsection 2.2.1. The `join` is the intersection defined in the In_n equation. We define how to track data using the `transfer` function in $Out_n = (Gen_n + In_n) - Kill_n(Gen_n + In_n)$. In the `transfer` function, we generate arithmetic expressions. We kill expressions in case a variable of that expression is modified. Dataflow equations also provide us with the means to compute a DFA automatically.

2.2.4 Worklist Algorithm

To automatically compute the result of a DFA, one can use a worklist algorithm. The worklist algorithm is a way to iteratively compute the result of the dataflow equations. Fig. 2.3 depicts the algorithm. In the algorithm, W is the worklist and F is the flow.

$Step1$ Initialization (of W and Analysis)
$W := nil;$
for all (l, l') in F do $W := cons((l, l'), W);$
for all l in F or E do
  if $l \in E$ then $Analysis[l] := \iota$ else $Analysis[l] = \bot_L$

$Step2$ Iteration (updating W and Analysis)
while $W \neq nil$ do;
  $l := fst(head(W)); l' = snd(head(W)); W := tail(W);$
  if $f_l(Analysis[l] \not\sqsubseteq Analysis[l'])$then
    $Analysis[l'] := Analysis[l'] \sqcup f_l(Analysis[l])$
    for all l'' with (l', l'') in F do $W := cons((l', l''), W);$

$Step3$ Presenting the result
for all l in F or E do
  $MFP_{entry}(l) := Analysis[l];$
  $MFP_{exit}(l) := f_l(Analysis[l]);$

Figure 2.3: Worklist Algorithm [3]

The flow consists of the edges of the CFG. The edges can be directed in either a forwards or a backwards manner. Further, E stands for the unique entry point of the analysis. ι specifies the initial or final analysis information. We specified it to be the empty set because our analysis starts with no calculated expressions. The transfer function is f_l. $f_l(Analysis[l])$ computes the information for the block l. Lastly, we still need the `join` from the dataflow equations. $\sqcup$ is the `join` and can either be an intersection or a union.

In the worklist algorithm, we find every part as defined in the dataflow equations. We start with all edges included in the CFG in the worklist. Going through the edges of the worklist, information is calculated as specified by the dataflow equations. Every time we calculate new information for an edge (l, l'), we add all possible edges (l', l'') to the worklist. If there is new information we have to propagate, we make sure that we pass it on. Thereby, we ensure that every time we add new information, we have a look if it influences any other statement. One can prove that the iteration reaches a fixed point, meaning the worklist algorithm stabilizes, terminates and computes the least solution [3].

Until now, we have presented a general DFA. Further, we provided the means to compute the result of a DFA automatically. Let us now have a look at a DFA described by Muske et al. in [6] to find equivalent warnings to remove redundant warnings.

2.3 Dataflow Analysis to Find Equivalent Warnings

The main idea of the DFA is similar to the *must* analysis presented in the last section. Now, the warnings become the information of interest. In case an expression throws a warning, we put it in the gen set. If an expression is in the gen set but not in the kill set, we set it active. We store all expressions that have at least one variable reassigned in the kill set and set them not active.

2.3.1 Example

Lst. 2.2 shows the program of Lst. 2.1 annotated with all warnings thrown. Using information about warnings we have, we can apply the DFA to find equivalent warnings.

```
1  void f1(int x, int y)
2  {
3    int h;
4    /*@ assert Value: division_by_zero: x ≠ 0; */
5    h = 1 / x;[1]
6    while (y)[2] {
7      /*@ assert Value: division_by_zero: x ≠ 0; */
8      h = 1 / x;[3]
9      x = y;[4]
10   }
11   /*@ assert Value: division_by_zero: x ≠ 0; */
12   h = 1 / x;[5]
13   /*@ assert Value: division_by_zero: x ≠ 0; */
14   h = 1 / x;[6]
15 }
```

Listing 2.2: Example Code with Redundant Warnings

We start with an empty set. In the first block, a warning is thrown. Thus, the gen set consists of the expression $1/x$. That set is propagated to the next block. In the

second block neither a warning is thrown nor a variable is reassigned, resulting in no changes. Another $1/x$ expression is generated in the third block. In the fourth block, x is reassigned. Therefore, we have to kill $1/x$, generated in the third block, again.

Having both paths that enter the second block, we can calculate the information in the second block. From the first block, we get the set with the expression $1/x$ and from the fourth block, we get the empty set. Intersecting these sets, since a warning has to be active on all paths, leaves us with the empty set. Since there is no expression active in the second block, we do not have to enter the `while loop` another time. Going on to the fifth block, we generate the expression $1/x$ again. In the sixth block, we see that the same expression throws a warning again. Since the expression throwing the warning was active before, we know that the warning is redundant. The result is depicted in Tab. 2.4.

Block	1	3	4	2	5	6
In	$\emptyset$	$\emptyset$	$\{1/x\}$	$\emptyset$	$\emptyset$	$\{1/x\}$
Out	$\{1/x\}$	$\{1/x\}$	$\emptyset$	$\emptyset$	$\{1/x\}$	$\{1/x\}$
Redundant Warning						✓

Table 2.4: DFA Result of the Warning Example

```
1  void f1(int x, int y)
2  {
3    int h;
4    /*@ assert Value: division_by_zero: x ≠ 0; */
5    h = 1 / x;[1]
6    while (y)[2] {
7      /*@ assert Value: division_by_zero: x ≠ 0; */
8      h = 1 / x;[3]
9      x = y;[4]
10   }
11   /*@ assert Value: division_by_zero: x ≠ 0; */
12   h = 1 / x;[5]
13   h = 1 / x;[6]
14 }
```

Listing 2.3: Example Code without Redundant Warnings

In this example, we could only remove one of the four warnings. However, that is still 25% of the warnings. Lst. 2.3 shows the result of the DFA without the reassignment in the fourth block.

- First block: The expression $1/x$ throws a warning. Thus, $1/x$ is now active.
- Third block: We set the expression $1/x$ active since it throws a warning.
- Second block: The intersection of the edges from the first and third block yields $1/x$ as active. Without the reassignment the expression $1/x$ survives the intersection.

- Fifth block: In the statement, the expression $1/x$, which is already active, throws another warning.
- Sixth block: The last statement also consists of the active expression $1/x$ throwing a warning.

Block	1	3	2	5	6
In	$\emptyset$	$\{1/x\}$	$\{1/x\}$	$\{1/x\}$	$\{1/x\}$
Out	$\{1/x\}$	$\{1/x\}$	$\{1/x\}$	$\{1/x\}$	$\{1/x\}$
Redundant Warning		✓		✓	✓

Table 2.5: DFA Result of the Warning Example without the Reassignment

Now the warning is active in every block after the first block as ca be seen in Tab. 2.5. Therefore, every but the first warning is redundant and could be removed. Without the reassignment we could remove 75% of the warnings.

2.3.2 Formal Definition

Until now, we explained the algorithm for the DFA by means of an example. Fig. 2.4 depicts the formal definition of the forwards dataflow equations [6]. The analysis is called *must* reaching expression analysis. It tracks equal-valued expressions to find redundant warnings. Hence, the expression must reach the current point in the program from every path. The equations use n for the corresponding node in the CFG. EOI stands for expression of interest. The EOI can be defined differently for different types of warnings. An example would be to only use the denominator for a devision-by-zero warning. However, we track the whole expression.

To understand the equations, one has to recall that statements are translated to nodes in a CFG. The entry of node n is called In_n and the exit Out_n. Let us show that the formal definition coincides with the explanation above. The first equation in Fig. 2.4 defines the entry of each block. Starting the DFA we cannot have any active warning. Hence, if we have the first block, we start with the empty set. Otherwise, we are in the CFG and we take the intersection of all the predecessors to calculate In_n. The data we have exiting a block is defined by what we had entering the block as well as what needs to be generated or killed during the block.

To know what needs to be generated, we need to have a look at the definition of Gen_n. The gen set contains the EOI if the current block has a warning. Otherwise, the gen set is empty. Therefore, we only set an expression as active if that expression throws a warning. Finally, we have a look at the kill set. In case no variable is modified or reassigned in the block, we kill nothing. Otherwise, we search for all the expressions that are active and contain any modified variables. The expressions returned by the $killInfo$ are all those with a modified variable. To find these we take the intersection of the modified variables with the variables used in each active expression. If the intersection

$$
\begin{aligned}
In_n &= \begin{cases} \emptyset & n \text{ is the entry block} \\ \bigcap_{p \in predecessors(n)} Out_n & \text{otherwise} \end{cases} \\
Out_n &= (Gen_n + In_n) - Kill_n(Gen_n + In_n) \\
Gen_n &= \begin{cases} \{e\} & n \text{ has a warning with } e \text{ as its EOI} \\ \emptyset & \text{otherwise} \end{cases} \\
Kill_n(X) &= \begin{cases} killInfo(X, n) & n \text{ modifies at least one variable} \\ \emptyset & \text{no variable is modified by } n \end{cases} \\
killInfo(X, n) &= \{e \in X \mid (usedVars(e) \cap modifiedVars(n)) \neq \emptyset\} \\
usedVars(e) &= \text{r-values from expression } e \\
modifiedVars(n) &= \text{l-values from program statement } n
\end{aligned}
$$

Figure 2.4: Forwards Dataflow Equations to Find Equivalent Warnings [6]

is not empty, we know that the expression contains the modified variable and we need to kill it. Concluding, the formal definition coincides with the informal explanation.

Next we can apply the equations to our CFG to see how these work. Starting with the first block, we get that the In_1 is the empty set since it is the entry block. In the first block, we have a warning. Therefore, the gen set contains the expression $1/x$. Now we have to check the kill set. The variable h is modified in the first block. Though, no expression of $Gen_1 + In_1$, which evaluates to $\{1/x\}$, contains h. Thus, the intersection of h and x is empty and we do not kill anything. At the exit of the first block we have, $Out_1 = \{1/x\}$

Assuming that In_3 is the empty set, we get that Out_3 is $\{1/x\}$ with the same argumentation as in the first block. In_4 is the intersection of all its predecessors. The fourth block has the third block as the only predecessor, leaving us with $Out_3 = In_4$. The fourth block has no warning, so the gen set is empty. However, we modify the variable x. The set modifiedVars contains the left side of the statement, namely x. Intersecting the set containing x with the variables of $\{1/x\}$ results in the non-empty set $\{x\}$. Therefore, $\{1/x\}$ is in the kill set. The set at the exit of block four is therefore $Out_4 = \{1/x\} - \{1/x\} = \emptyset$.

Now we can calculate In_2. It is the intersection of Out_1 and Out_4. Intersecting $\{1/x\}$ with $\emptyset$ leaves us with $\emptyset$. The second block has no warning and we do not modify any variable. Therefore, we have $In_2 = Out_2 = \emptyset$. Knowing that Out_2 is the empty set, we also know that our assumption for In_3 holds and that we do not have to enter it again.

The fifth and sixth block are still not calculated. Out_2 defines In_5 that is the empty

set. In the fifth block we generate the expression $\{1/x\}$ again and kill nothing. In_6 is equal to Out_5 since it is its only predecessor. In block six, we generate $\{1/x\}$ again but since it already is in the active set, it is not added again. Further, we do not kill anything. Therefore, we know that we can group the warning from block 5 and 6 together. Furthermore, the warning from the fifth block has a path to the warning in the sixth block, but not the other way around. Earlier we defined the most general warning as the one having a path to all others. Knowing that the warning from the fifth block is the more general one, we can remove the warning in the sixth block. Hence, we get the same result as depicted in Lst. 2.3. Tab. 2.6 depicts the application of the dataflow equations to our example from Fig. 2.1. Further, the results depicted in Tab. 2.4 and Tab. 2.6 are the same. Therefore, the informal and the formal definition compute the same result.

Block	1	3	4	2	5	6
In	$\emptyset$	$\emptyset$	$\{1/x\}$	$\emptyset$	$\emptyset$	$\{1/x\}$
Out	$\{1/x\}$	$\{1/x\}$	$\emptyset$	$\emptyset$	$\{1/x\}$	$\{1/x\}$
Redundant Warning						✓

Table 2.6: Dataflow Equations Applied to Our Example

Now, we know the theoretical background to find equivalent warnings and how to apply the algorithm automatically to our source code. In the next chapter, we use that knowledge and take a look at the implemented plug-in and a few more examples.

3 Plug-In to Remove Redundant Warnings

Frama-C provides functionality to statically analyze C code. Further, it helps developers to implement a DFA. For example, the developer does not have to implement how the `transfer` function is applied in the analysis. Therefore, the *must* reaching analyses is implemented using the Frama-C framework. In this chapter, we first take a closer look at Frama-C. After knowing how Frama-C can help us, we explain our implementation. Lastly, we ensure that the implementation works as expected and evaluate it.

3.1 Frama-C

Frama-C is an open-source, platform–independent framework to statically analyze C code. The whole framework is written in OCaml. OCaml is a functional programming language. It has influenced languages such as F# and Scala.

The Frama-C framework consists of different plug-ins for different tasks and analyses. These plug-ins can be used in combination to analyze C code [2]. In the process of analyzing C code, these plug-ins annotate the Frama-C internal representation of the C code, namely CIL. The annotations have a unique id to distinct them. Further, they consist of what the plug-ins could conclude analyzing the code. The plug-ins can see what the other plug-ins annotated and work with that information. Thereby, one can call different plug-ins in combination. Further, Frama-C provides the means to register new plug-ins. A newly written plug-in can then be combined with already existing plug-ins.

One of the probably most widely used Frama-C plug-in is the value analysis. The value analysis plug-in is a *correct* static analyzer [5]. *Correct* means that if anywhere at run time a bug can occur, it warns about it. The value analysis can be used to verify C code against provided specifications. In case one does not provide any specifications, the value analysis only tries to find common run-time bugs, e.g., division-by-zero or array-index-out-of-bounds. To analyze the code, it is transformed into the CIL intermediate representation(IR). If the value analysis or many other plug-ins find a possible bug, they annotate the IR with a warning.

The framework also provides a number of modules with functionalities to help the user implement their own plugin. One of the modules provides functions for either a forwards or backwards DFA. To use that module, one only has to define the `transfer` function and the `join`. A function to implement the `join` is how to combine predecessors. After we specify these functions, Frama-C applies the DFA to the C code. While applying the DFA, Frama-C uses a worklist algorithm to compute a solution to the dataflow equations.

3.2 Implementation of the Plug-In

As Frama-C is written in OCaml, the new plug-in is also written in Ocaml. We implemented the plug-in using Frama-C Fluorine-20130601 on a Mac OSX 10.8 with Ocaml

version 4.00.1. To implement the DFA described by Muske et al. in [6], the plug-in needs to know the expressions that throw a warning. As aforementioned, Frama-C plug-ins such as the value analysis can annotate the internal representation with the warnings they threw. Therefore, we first need to be able to go through that IR and retrieve all the warnings other plug-ins annotated. To that end, Frama-C provides the `in-place visitor` module. Such an `in-place visitor` can be used to go through the IR. We now have to collect all the annotations that are warnings from the IR. Collecting the annotations, we also get the information which statement the annotation belongs to.

Having the prerequisites for the analysis, we can start to implement it. To implement the analysis we use the `Dataflow` module of Frama-C. The `Dataflow` module provides a module for a forwards analysis and a backwards analysis. We are interested in the module for a forwards analysis. To use the module, all its functions need to be specified. That means we need to provide the implementation for every function of the module. The functions that we have to specify correlate to the `join` and `transfer` functions. Further, we need to specify our flow data. Flow data are our information of interest. However, the flow data consists not only of a set of expressions but also of a set of annotations that can be removed. In case a statement throws a warning, we save everything to the right of the equal sign as expression. That means that if $h = 1/x$ throws a warning we save the expression $1/x$. In case $b = a[1/x]$ throws a warning, we save the expression $a[1/x]$. Now, we can start with the analysis. Initially, we pass the analysis all the annotations that belong to warnings.

The most interesting functions for us are `combinePredecessors` and `doInstr`. In the `combinePredecessors` function, the `join`, we specify that we have a *must* and not a *may* analysis. Therefore, we use the intersection on the expressions we track in the `combinePredecessors` function. The `transfer` function is implemented in the `doInstr` function. Every time we see a `set instruction` ,e.g., `h=x`, we go through the flow equations from Fig. 2.4. First of all, the plug-in checks the annotations retrieved early and passed to the analysis if we find an annotation corresponding to the statement of the instruction. If the plug-in finds an annotation it knows the expression of the statement throws a warning. Otherwise, the statement does not throw any warning. If the plug-in finds any annotation that relates to the statement, the expression of the statement makes up the gen set that is then added to the flow data.

For the kill set, the plug-in has to check the left value, `lvals`, of every instruction. Knowing the `lval` , the plug-in "kills" every expression of the flow data that contains the `lval`. That is exactly what the flow equations define. An expression is killed by removing it from the flow data. After removing the expression it is not active anymore. Next we need to find redundant warnings. In case an expression throws a warning and is not active, we set the expression active by adding it to our set of expressions in the flow data, but leave the annotations unchanged. If the expression is active and the expression throws a warning, we do not add the expression again, but add the annotations to the to-be-removed set. However, it can be the case that we later realize the warning was not redundant. Then, we have to remove the annotation from the to-be-removed set We only add annotations of already active expressions to it. Thereby, we ensure that the first warning with a path to the other warnings is the one being displayed.

Now we have the result of the DFA and the warnings that can be removed. The last step is to remove the warnings. For this, we have to go through the IR again. In contrast to the first walkthrough, we do not retrieve all the warnings, but remove the warnings that we classify redundant. Each annotation has a unique id. While the plug-in goes through the IR it checks for the ids of annotations we classified as redundant. In case the plug-in finds a corresponding annotation in the IR it removes it. In the next section we take a look at how the plug-in performs.

3.3 Evaluation

We start the evaluation by having a look at how the plug-in calculates the result for the annotated code given in Lst. 2.2. Afterwards, we have a look at the properties that have to hold for the plug-in. Lastly, we evaluate the plug-in by testing it

3.3.1 Example

First of all, the plug-in goes through the IR and collects the four annotated warnings. The plug-in passes the warnings to the analysis and starts the analysis using the worklist algorithm to compute the solution. In the beginning, the set of flow data is empty. Therefore, no expressions are active when the analysis starts and no warnings are to be removed. The worklist contains every edge of the CFG from Fig. 2.1. The first `set instruction` is `h = 1 / x;` from the first block. Now the plug-in has to go through the flow equations from Fig. 2.4. First, it calculates the gen set. Hence, the plug-in has to check if the statement throws a warning.

To check if a statement throws a warning the plug-in needs to access the information about the warnings passed in the beginning. For our statement, it sees that a warning corresponds to that statement. Thus, it can add the expression $1/x$ of the statement to the gen set. Afterwards, the plug-in has to calculate the kill set. To calculate the kill set, the plug-in has to take the `lval` of the statement `h = 1 / x;` and intersect it with the variables of the active expressions. The `lval` of `h = 1 / x;` is h and intersecting that with the only variable in the active expressions x, we get an empty set. Since the intersection is empty, our kill set is empty as well. Therefore, the flow data consist of the expression $1/x$ after the first block.

The worklist algorithm propagates that information onwards to the next block. Since the information on the edge $(1, 2)$ changes from the empty set to $\{1/x\}$, the plug-in adds the edges $(2, 3)$ and $(2, 5)$ to the worklist. In the second block, it does not have to do anything at the moment. Going on with the edge $(2, 3)$, the plug-in enters the `while loop` and has the expression $1/x$ in the flow data. The third block has the `set instruction` `h = 1 / x;`. Further, the statement has a corresponding warning. Putting the expression $1/x$ in the gen set, the plug-in sees that the expression is already active. Therefore, it marks that annotation as redundant by adding to the to-be-removed set in the flow data. For the same reason as in the first block, the kill set is empty in the third block. Taking the next edge to the fourth block, the flow data still consists of the expression $1/x$.

The `set instruction` of the fourth block is `x = y;`. There is no corresponding warning to that statement. Therefore, the gen set is empty. The `lval` of the expression is x. Intersecting that x with the variable of $1/x$, we get x. Thus, the kill set consists of $1/x$. Removing the expression $1/x$ from the flow data results in an empty set.

Entering the second block again, we apply the `combinePredecessors` function. From the first block, the plug-in gets the information that the flow data consist of the expression $1/x$ and from the fourth, that the flow data is empty. Intersecting the sets of information, it gets that no expressions are in the flow data entering the second block. Now, the plug-in has to add the edges $(2, 3)$ and $(2, 5)$ to the worklist again to propagate the information that actually no expressions are in the flow data.

Going into the `while loop` a second time, the plug-in sees again that it has $1/x$ in the gen set. This time, there is no active expression. Therefore, the expression is put in the gen set. However, the plug-in marked that warning as redundant before, but it now knows that this warning is not redundant. Hence, it has to remove that warning from the set of redundant warnings in the flow data. The rest is exactly the same as explained above for the `while loop`. Entering the second block again, the plug-in applies the `combinePredecessors` function. It gets the same result as before and does not have to propagate anything new. The next edge in the worklist is the edge $(2, 5)$. Entering the fifth block, the plug-in has no expression active. The gen set of the fifth block consists of $1/x$ and the plug-in does not kill anything. Leaving the fifth block the expression $1/x$ is in the flow data and the plug-in propagates it to the sixth block. The sixth block has the `set instruction h = 1 / x;` with a corresponding warning. The expression $1/x$ from the gen set is already active and thus the plug-in saves the warning as redundant.

Step	Block	Gen Set	Kill Set	Expressions	Warnings to be Removed
1	1	$1/x$	$\emptyset$	$1/x$	$\emptyset$
2	2	$\emptyset$	$\emptyset$	$1/x$	$\emptyset$
3	3	$1/x$	$\emptyset$	$1/x$	Warning 2
4	4	$\emptyset$	$1/x$	$\emptyset$	Warning 2
5	2	$\emptyset$	$\emptyset$	$\emptyset$	Warning 2
6	3	$1/x$	$\emptyset$	$1/x$	$\emptyset$
7	4	$\emptyset$	$1/x$	$\emptyset$	$\emptyset$
8	2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
9	5	$1/x$	$\emptyset$	$1/x$	$\emptyset$
10	6	$1/x$	$\emptyset$	$1/x$	Warning 4

Table 3.1: Example Computation using the Plug-In

Tab. 3.1 shows each step of computation. The set of expressions and warnings to be removed are the flow data at the exit point of the block. Now, we have to remove the fourth warning. To remove the warning, the plug-in goes through the IR. When it finds the annotation with the same id as the one to be removed, it removes that annotation from the IR. Frama-C now displays exactly what we see in Lst. 2.3.

3.3.2 Properties of the Tool

Let us have a look at the properties of the plug-in. First of all, we need another Frama-C plug-in to annotate the IR with warnings. We use the value analysis to annotate the warnings. However, our plug-in should work with every other Frama-C plug-in that annotates the IR with warnings. Further, the plug-in checks the whole expression that throws a warning for structurally equality to group them. It also allows a combination of warnings to make up the most general warning.

According to the aforementioned definition of EOI, we use the whole expression to track it. Muske et al. track only the denominator for a division-by-zero or the index of an array for an array-index-out-of-bound(AIOB) warning. However, one has to save what kind of warning the EOI referred to. For the division-by-zero that approach should work well. Saving only the index for an array for AIOB warning can lead to a problem:

```
1  void f1 (int x) {
2    const int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
3    const int b[] = {0, 1, 2, 3, 4};
4    /*@ assert Value: index_bound: 0 ≤ x; */
5    /*@ assert Value: index_bound: x < 10; */
6    int h = a[x];
7    /*@ assert Value: index_bound: x < 5; */
8    int h = b[x];
9  }
```

Listing 3.1: Array Example Code

The example of Lst. 3.1 illustrates the problem. In case we only save the x as EOI, we might group these warnings together. However, we access arrays of different sizes. Removing the second warning would be unsound, but is possible by following the flow equations of Fig. 2.4. Removing the second warning, we would run into problems if $x = [5, 9]$. In that case, we would conclude that the first warning is a false positive, but the second warning's fault occurs. However, since the first is a false positive we would not look at the second warning.

If we would only save the denominator for a division-by-zero and the index for an AIOB, we could not differentiate between $a[x]$ and $1/x$ since we would save x in both cases. Saving the whole expression, however, ensures that we only compare warnings thrown by exactly the same expression. We also do not have to track what kind of warning it is since we save the whole expression. Nevertheless, there are also a few drawbacks. If the array length is the same, it would be nice to group the warnings together if they have the same index. On the other hand the value analysis of Frama-C is already good when it comes to AIOB warnings. If two arrays have the same length, the value analysis only throws one warning. A second drawback, in the implementation we compare expressions to be structurally equal. The comparison yields that $\frac{1}{x} \neq \frac{2}{x}$. Therefore, two division-by-zero warnings end up in different baskets if they have different numerators. However, if two warnings are in the same set, we are sure that they belong together. Hence, we use the whole expression.

There is another point in which the implementation differs from the description from Muske et al. in [6]. The code in Lst. 3.2 has an `if` condition. In both branches, a warning is thrown and we set the expression $1/x$ active. Combining the branches, we take the intersection of both branches. Since $1/x$ is active on both branches taking the intersection of it gives us $1/x$. When we reach the warning after the `if` block the expression $1/x$ is active. Therefore, applying our tool removes the last warning. Muske et al. require for each redundant set one most general warning. In our case, we allow a warning in each branch to combine to the most general warning.

```
1  void f1(int x) {
2    int h;
3    if (x != 0)
4      /*@ assert Value: division_by_zero: x ≠ 0; */
5      h = 1/x;
6    else
7      /*@ assert Value: division_by_zero: x ≠ 0; */
8      /*@ assert Value: signed_overflow: 1/x ≤ 2147483647; */
9      /*@ assert Value: signed_overflow: -2147483648 ≤ 1/x;*/
10     h = 1/x;
11   /*@ assert Value: division_by_zero: x ≠ 0; */
12   h = 1/x;
13 }
```

Listing 3.2: Example Code for Combing Branches

Combining the branches, we have the same predicate that has to hold in each branch. Showing the warnings in the branches, we have to make sure that the predicate holds in every branch. If it holds in every branch, it also holds after combining the branches. Therefore, we can remove the warning after the `if` block with the same predicate. Warnings thrown by the same expression produce annotations with same predicates. Further, displaying the warnings in the branches also works if x is reassigned in a branch and afterwards, the same expressions throws a warning again. In this case, we still have the same predicate in each branch and if it holds in each branch it also holds after the `if` block. Hence, as long as we have the same predicate in every branch, we can assume that it also have to hold after the combination of the branches. Having an idea of how the plug-in works and it properties, we still need to see how it performs.

3.3.3 Evaluation Against a Built-In Function

To see how the implemented plug-in performs, we compare the output of the plug-in against the output of a built-in plug-in of the value analysis. The value analysis has a not very well known option `-remove-redundant-alarms` [1]. The option has the goal to remove redundant alarms. Therefore, it has the same goal as our analysis, but it only works in combination with the value analysis. From what we understand, the remove redundant alarms option is tracking the annotations and the modification point of that annotation. Further, if two annotations have the same modification point, the built-in

function compares the predicates of the annotations. That means if two annotations have the same modification point and both have a warning with the same predicate, let us say $x \neq 0$, then they remove a warning. They analyze the code in a forward manner.

Now, we can provide both plug-ins with the same C code to analyze and compare the output. While we apply different examples to the plug-ins and compare the outputs of both analyses, we can see that they return nearly the same result. However, there are a few differences that can be explained by the way the warnings are tracked. The built-in option groups warnings thrown by expressions like $1/x$ and $2/x$ together because the annotations have the same predicate. The implemented plug-in does not group them together because it tests on structural equality for the whole expression. Therefore, the built-in function can remove a few more warnings and is more precise. Lst. 3.2 contains another example where the output of the plug-ins differ. The built-in function is not able to remove the warning corresponding to line 12. As aforementioned, the plug-in does remove that warning and is more precise in this case.

```
1  void f1(int x) {
2    int h;
3    /*@ assert Value: division_by_zero: x ≠ 0; */
4    h = 1/x;
5    if (x != 0)
6      /*@ assert Value: division_by_zero: x ≠ 0; */
7      h = 1/x;
8    else
9      /*@ assert Value: division_by_zero: x ≠ 0; */
10     /*@ assert Value: signed_overflow: 1/x ≤ 2147483647; */
11     /*@ assert Value: signed_overflow: −2147483648 ≤ 1/x;*/
12     h = 1/x;
13 }
```

Listing 3.3: Too Many Warnings Example Code

The last difference is that sometimes an expression throws additional signed-overflow warnings in a branch, due to over-approximation. An example of that shows Lst. 3.3. The plug-in removes these additional warnings if it has a redundant warning for that expression. In case an expression is active and throws a warning again, the plug-in adds all annotations of that expression to the to-be-removed set. Therefore, it removes these warnings. The built-in plug-in cannot remove the warnings because they have other predicates and are not redundant to another warning. The expression probably should not have thrown these additional warnings to start with. In the `else` branch, the valuation of x can only be restricted That means that the x in the `else` branch cannot obtain any value that is not possible before the `if` block. Hence, either that warning should be also displayed before the `if` block or should not be possible in the branch. The outputs of both plug-ins are correct, we, however, are more precise. We reported the superfluous signed-overflow warnings and they will not appear in the next version of the value analysis anymore.

3.3.4 Evaluation of Different Combinations

So far in this chapter, we have provided an overview of the implementation and compared it to a built-in plug-in. The results of the comparison are promising. Both plug-ins return about the same result. Further, the few differences can be explained by the implementation. Furthermore, the outputs of both plug-ins are sound, they only differ in precision.

In this subsection, we want to evaluate the correctness of the implemented plug-in. To evaluate it, we conduct a case study. In the case study, we test different possible combinations of C statements. Tab. 3.2 and Tab. 3.3 depict the combinations tested. The statements are ordered as they occur in the code. In case a warning appears in a `while loop`, we attach that warning to the loop. In terms of labeling that means if a `while loop` is the first statement, then it has the number 1. If a warning is the first statement in the `while loop`, we label it with 1.1 and so on.

Statement	Order					
Warning	1, 2	1, 2, 4, 5	1.1, 2.1, 3	1.1.1, 1.2.1	1, 2.1, 3.1, 4	1, 2.1
Reassignment		3				2.2
If			1	1.1	2	
Else			2	1.2	3	
While				1		2
Works	✓	✓	✓	✓	✓	✓

Statement	Order					
Warning	1, 2.1	1, 2.1, 2.3	1, 2.1	1, 2.1, 3	1, 2.1, 3.2, 4	1.1, 2.1, 3
Reassignment	2.2	2.2			3.1	
If					2	1
Else					3	2
While	2	2	2	2		
Works	✓	✓	✓	✓	✓	✓

Table 3.2: Combinations of Statements

Let us first go through a few combinations to understand the notation. In Tab. 3.2, we have two tables, each with six different combinations. The first combination of the first table has two sequential statements. A warning is thrown in the first statement, followed by a statement throwing the same warning. The implemented plug-in removes the second warning correctly. Hence, in the Tab. 3.2 we mark the combination as working.

The next combination is a test on reassignment. We have two statements with warnings followed by a reassignment. Afterwards, there are two more statements containing warnings. The plug-in removes the second and fourth warning correctly. We have a reassignment before the third warning. Therefore, the expression corresponding to the warnings is not in the flow data anymore and the plug-in does not remove the third warning. Looking at the output of the plug-in for this combination, we see the expected result. Thus, we mark that combination as working.

Until now, we had no flow altering statements. The next combination has an `if` condition, which alters the flow, as first statement. We split up the `if` block into the `if` branch, which is actually the `then` branch, and the `else` branch. Therefore, the first statement is the `then` branch of the `if` block. In the `then` branch, the first statement includes a warning, denoted by 1.1. In case we had another statement in the `then` branch, it would get the label 1.2. The `else` branch is the next statement and contains one warning. After the `if-then-else` block, we have another warning in the last statement. The plug-in removes the last warning correctly, as explained in subsection 3.3.2 when we looked at the example in Lst. 3.2.

Statement	Order	
Warning	1.1.1, 1.2.1, 2.1, 3.1, 4.1, 5.1	1, 2.1.1, 2.2.1, 3.1, 4.1, 6.1, 7.1
Reassignment		5
If	1.1, 2, 4	2.1, 3, 6
Else	1.2, 3, 5	2.2, 4, 7
While	1	2
Works	✓	✓

Statement	Order	
Warning	1.1, 1.2.1, 1.3.1, 2.1, 3.1, 4.1, 5.1	1, 2.1.1, 2.2.1, 3.1, 4.1, 6.1, 7.1
Reassignment		2.1.2, 5
If	1.2, 2, 4	2.1, 3, 6
Else	1.3, 3, 5	2.2, 4, 7
While	1	2
Works	✓	✓

Table 3.3: Combinations of Statements

We test different combinations of warnings and reassignments composed with statements that can alter the flow. All of these tests return the expected result. Further, we include only the `while loop` but not also the `for loop` as a statement because the `for loop` gets translated into a `while loop` internally, at least in the value analysis of Frama-C. Mainly, we test that an expression has to be active on every path. More precisely, we test that we handle loops and their fix point iteration correctly. Let us now have a look at different combinations.

The code in Lst. 3.4 combines a number of flow changing statements with warnings and reassignments. It correlates to the fourth combination in Tab. 3.3. However, the code in Lst. 3.4 is missing the last `if` block compared to the fourth combination in Tab. 3.3. Applying the implemented plug-in to that code, we see that the plug-in cannot remove any warnings. The plug-in first sets $1/x$ active before the `while loop` and enters it with $1/x$ active. In the `while loop`, the plug-in first removes the warnings in each branch. However, since we reassign x in one branch, $1/x$ is not active anymore after combining the `if` branches. Combining the edge from the `while loop`, which has no warnings active, and the one from line 6, we get that no warnings are active. Now, we have to go through the `while loop` again. In the third block, the plug-in sees that it should not

have removed the warnings and adds them again by removing the annotation from the flow data. We reach the `if` condition in line 20 with no warnings active. Therefore, we cannot remove any of the warnings in the branches. After combining the branches, $1/x$ is active but no warning follows.

```
1  void f1(int x){
2    int b, y;
3    b = 1;
4    /*@ assert Value: division_by_zero: x ≠ 0; */
5    y = 1 / x;
6    while (b < 10) {
7      if (x != 0) {
8        /*@ assert Value: division_by_zero: x ≠ 0; */
9        y = 1 / x;
10       x = 1;
11     }
12     else {
13     /*@ assert Value: division_by_zero: x ≠ 0; */
14     /*@ assert Value: signed_overflow: 1/x ≤ 2147483647;*/
15     /*@ assert Value: signed_overflow: -2147483648 ≤ 1/x;*/
16       y = 1 / x;
17     }
18     b ++;
19   }
20   if (x) {
21     /*@ assert Value: division_by_zero: x ≠ 0; */
22     y = 1 / x;
23   }
24   else {
25     /*@ assert Value: division_by_zero: x ≠ 0; */
26     /*@ assert Value: signed_overflow: 1/x ≤ 2147483647; */
27     /*@ assert Value: signed_overflow: -2147483648 ≤ 1/x;*/
28     y = 1 / x;
29   }
30 }
```

Listing 3.4: Combination of Statements

In case we removed the reassignment in line 11, the implemented plug-in would remove nearly all warnings. The combination correlates to the second combination in Tab. 3.3. Without the reassignment, $1/x$ would still be active after combining the `if` branches. Therefore, $1/x$ would also be active when we combine the predecessors of the `while loop` and we would not have to enter the `while loop` again. Now, $1/x$ is active reaching the `if` condition in line 20. Here, we remove the warnings in the branches of the `if` block. Consequently, the plug-in would remove every warning but the very first. In case the very first warning is a false positive, the reviewing effort would be reduced drastically.

4 Future Work & Conclusion

4.1 Future Work

The whole approach by Muske et al. consists of a combination of backwards analysis and forwards analysis to find redundant warnings. So far the plug-in implements the forwards analysis, but the backwards analysis is missing. Combining these analyses, one can find larger sets of redundant warnings. Therefore, fewer warnings need to be displayed which could reduce the reviewing efforts even more.

Further, using Frama-C it might be better to use annotations instead of expressions. Tracking annotations, we can compare them by their predicates. Using these predicates, we probably can group more annotations together as we do now. However, it could be harder to find the annotations we need to kill without saving the expressions.

```
1  void f1(int x) {
2    int h;
3    if (x != 0)
4      h = 1/x;
5    else
6      h = 1/x;
7  }
```

Listing 4.1: Path Sensitivity Example

Having the analysis as described by Muske et al., there is still room for improvements. We could add some path sensitivity to the analysis. Lst. 4.1 is an example where path sensitivity can improve the result of the analysis. In case the condition of the `if` holds, we know that the expression in line 4 cannot be a division-by-zero fault. Here, we can propagate that $x \neq 0$ to the `then` branch. Therefore, we do not have to display a division-by-zero warning in the `then` branch unless we reassign x to 0,. Further, we could try to track assignments.

```
1  void f1(int x, int y) {
2    int h = 1/x;
3    y = x;
4    h = 1/y;
5  }
```

Listing 4.2: Assignment Tracking Example

Lst. 4.2 shows an example where assignment tracking could help us. Tracking that `y=x`, we can conclude that the warnings corresponding to line 2 and 4 are actually redundant. Still, even though there are a few topics open, the plug-in already is a working prototype.

4.2 Conclusion

The goal of this project has been to reduce the efforts of reviewing warnings. To that end, a plug-in for Frama-C was implemented. The plug-in is based on the forwards analysis described by Muske et al. in [6]. For the plug-in to work, it needs another Frama-C plug-in to annotate the IR with warnings first. To remove redundant warnings, the plug-in finds equal-valued expressions that throw warnings using a DFA. Having the set of equal-valued expression and the corresponding warnings, the plug-in has to display the most general warning for each set of equal-valued expression.

We tested the plug-in, to ensure that we only remove warnings that can be removed. To be more precise, we handle the fix point finding for loop iterations correctly. By displaying fewer warnings, while still providing the same information, the reviewing effort of the user is reduced in most cases. Therefore, using the plug-in can be of great help for the user.

Bibliography

[1] Cuoq, P. Minimizing-alarms. `http://blog.frama-c.com/index.php?post/2012/03/12/Minimizing-alarms`, Jan. 2014. Accessed: 2014-01.

[2] Cuoq, P., Signoles, J., Baudin, P., Bonichon, R., Canet, G., Correnson, L., Monate, B., Prevosto, V., and Puccetti, A. Experience report: Ocaml for an industrial-strength static analysis framework. *SIGPLAN Not. 44*, 9 (Aug. 2009), 281–286.

[3] Flemming Nielsen, H. N., and Hankin, C. *Principles of Program Analysis*, 2nd ed. Springer, 2005.

[4] Kildall, G. A. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1973), POPL '73, ACM, pp. 194–206.

[5] Loïc Correnson, Pascal Cuoq, F. K. V. P. A. P. J. S., and Yakobowski, B. Frama-C user manual. `http://frama-c.com/download/frama-c-user-manual.pdf`. Accessed: 2014-01.

[6] Muske, T. B., Baid, A., and Sanas, T. Review efforts reduction by partitioning of static analysis warnings. In *IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2013* (2013), IEEE, pp. 106–115.

Appendix

```
(** combine two branches.
  Take the intersection of the expressions -> MUST
  Take the union of the warnings -> all need to be removed,
     to be able to remove an annotation the third list is
     added *)
let combinePredecessors _ ~old new_ =
    let (a,b,c) = old in
    let (d,e,f) = new_ in
    if equal (a,b,c) (d,e,f) then None else Some ((
       Cil_datatype.ExpStructEq.Set.inter  a d),(
       Cil_datatype.Code_annotation.Set.union b e), (
       Cil_datatype.Code_annotation.Set.union c f))
```

Listing 3: combinePredecessors

```
  (** actual forwards transfer function for our analysis.
  for each instruction we match the instruction if it is a
     set operation (a=b;).
  In case of a set operation we go on with our analysis and
     see if a warning is involved and so on.
  So far, a call is not implemented. Therefore, we can not
     handle a call of a different function by now.
  Assemebly is also not supported.
  *)
let doInstr _stmt instr flowval =
     match instr with
       | Set (lval, exp, _) ->
              Dataflow.Done (calculate_next_flow_data lval
                 exp flowval Params.myWarnings _stmt)
       | Call _ -> Self.warning "Function call not
          implemented - setting \to unsafe"; Dataflow.
          Default
       | Asm _ -> Self.warning "Assembler? Really?";
          Dataflow.Default
       | _ -> Self.result "Instr: %t " (Extlib.swap
          Cil_datatype.Instr.pretty instr);
              Dataflow.Default

  (** returns all variables of an exp *)
let rec find_lvals exp = match exp.enode with
    | Lval lval -> [lval]
```

```
19      | SizeOfE exp -> find_lvals exp
20      | AlignOfE exp -> find_lvals exp
21      | UnOp (_, exp, _) -> find_lvals exp
22      | BinOp (_, exp1, exp2, _) -> find_lvals exp1 @
          find_lvals exp2
23      | CastE (_, exp) -> find_lvals exp
24      | AddrOf lval -> [lval]
25      | StartOf lval -> [lval]
26      | Info (exp, _) -> find_lvals exp
27      | _ -> []
28
29 (** Here we check if in the current statement either a
      warning is thrown, in that case we add it to our list of
      expressions, if it a new exp otherwise we add it to the
      redudandant warnings, or if a variable of any exp is newly
       assigned in that case we kill all expressions with that
      variable. *)
30 let calculate_next_flow_data lval exp (flowval,warn,removes)
      myWarnings _stmt =
31   let is_warning = is_there_current_warning myWarnings _stmt.
       sid in
32   let (new_flow,new_warn,new_removes) = match is_warning with
33     | false -> (flowval,warn,removes)
34     | true when (exp_exists_in_flowdata flowval exp) -> (
         flowval, insert_warning warn (get_current_code_annot
         myWarnings _stmt.sid),removes)
35     | true when check_for_annots (get_current_code_annot
         myWarnings _stmt.sid) warn -> (Cil_datatype.
         ExpStructEq.Set.add exp flowval,warn, insert_warning
         removes (get_current_code_annot myWarnings _stmt.sid))
36     | true -> (Cil_datatype.ExpStructEq.Set.add exp flowval,
         warn,removes)
37   in
38   let kill_set = check_for_lval (Cil_datatype.ExpStructEq.Set
       .elements new_flow) lval in
39   (remove_flowdata kill_set new_flow,new_warn,new_removes)
40
41 (**  returns true of for a given expression it contains a
      certain variable.*)
42 let rec lval_is_in_exp exps lval = match exps with
43   | [] -> false
44   | x::_ when Cil_datatype.LvalStructEq.equal x lval -> true
45   | (_, Index (e, _))::_ when lval_is_in_exp (find_lvals e)
       lval -> true
```

```
46   | _::xs -> lval_is_in_exp xs lval
47
48 let lval_in_exp exp lval =
49   let lvals_of_exp = find_lvals exp in
50   lval_is_in_exp lvals_of_exp lval
51
52 (** check if a variable is contained in an expression and
       contain a list of all exp that contain that variable. *)
53 let rec check_for_lval flowval lval = match flowval with
54   | [] -> []
55   | x::xs when lval_in_exp x lval -> x :: check_for_lval xs
       lval
56   | _::xs ->   check_for_lval xs lval
57
58 (** remove the expressions from the flow data, in which a
       newly assigned variable is *)
59 let rec remove_flowdata kill_set flowval = match kill_set
     with
60   | [] -> flowval
61   | x::xs -> (*Self.result "kill: %a" Cil_datatype.
       ExpStructEq.pretty x;*) remove_flowdata xs (Cil_datatype
       .ExpStructEq.Set.remove x flowval)
62
63 (** insert a list of annotations in a set of annotations *)
64 let rec insert_warning old new_ = match new_ with
65   | [] -> old
66   | x ::xs -> insert_warning  (Cil_datatype.Code_annotation.
       Set.add x old) xs
67 (** remove a warning from the set *)
68 let rec remove_warning old new_ = match new_ with
69   | [] -> old
70   | x ::xs -> remove_warning  (Cil_datatype.Code_annotation.
       Set.remove x old) xs
71 (** check if annotation is in the set *)
72 let rec check_for_annots annots flow = match annots with
73   | [] -> false
74   | x::_ when Cil_datatype.Code_annotation.Set.mem x flow ->
       true
75   | _::xs ->   check_for_annots xs flow
```

Listing 4: doInstr